<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *International Symposium on VLSI Design (ISVLSI)*, Miami, Florida, USA, July 15-17, 2019.

Citation for the original published paper:

Savas, S., Yassin, A., Nordström, T., Ul-Abdin, Z. (2019)
Using Harmonized Parabolic Synthesis to Implement a Single-Precision Floating-Point Square Root Unit
In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (pp. 621-626).
IEEE conference proceedings
VLSI, IEEE Computer Society Annual Symposium on
<https://doi.org/10.1109/ISVLSI.2019.00116>

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-39322>

Using Harmonized Parabolic Synthesis to Implement a Single-Precision Floating-Point Square Root Unit

Süleyman Savas, Yassin Atwa*, Tomas Nordström, Zain Ul-Abdin
School of Information Technology, Halmstad University, Halmstad, Sweden

{firstname.lastname}@hh.se, *yasatw13@student.hh.se

Abstract—This paper proposes a novel method for performing square root operation on floating-point numbers represented in IEEE-754 single-precision (binary32) format. The method is implemented using Harmonized Parabolic Synthesis. It is implemented with and without pipeline stages individually and synthesized for two different Xilinx FPGA boards.

The implementations show better resource usage and latency results when compared to other similar works including Xilinx intellectual property (IP) that uses the CORDIC method. Any method calculating the square root will make approximation errors. Unless these errors are distributed evenly around zero, they can accumulate and give a biased result. An attractive feature of the proposed method is the fact that it distributes the errors evenly around zero, in contrast to CORDIC for instance.

Due to the small size, low latency, high throughput, and good error properties, the presented floating-point square root unit is suitable for high performance embedded systems. It can be integrated into a processor's floating point unit or be used as a stand-alone accelerator.

I. INTRODUCTION

Floating-point square root is a significant, fundamental arithmetic operation used by a vast number of applications including radar and baseband signal processing. For instance, QR decomposition [1]–[4], which is one of the major factorizations in linear algebra and used in Massive MIMO [5], performs significant amount of square root operations on floating-point numbers. However, it is quite challenging to implement efficient floating-point arithmetic in hardware. Square root itself is one of the most challenging basic operations to implement among the others such as addition, multiplication, and subtraction. It requires larger area and usually achieves relatively lower performance. Additionally, aforementioned operations are computed as approximations, which introduces errors in the results. These errors may accumulate and cause accuracy degradation in the final result unless the errors are evenly distributed around zero.

In this paper, we implement a single-precision square root hardware that performs square root on floating-point numbers represented in IEEE-754 standard format [6]. The implementations are realized and tested on two field-programmable gate arrays (FPGA) for validation and evaluation.

The proposed method relies on approximation that adopts Harmonized Parabolic Synthesis [7], [8]. It converges faster and provides faster computation and smaller chip area when

compared to the other methods such as CORDIC [9], [10] and Newton Raphson [11], [12]. Additionally, in contrast to the other methods, the distribution of the error can be tailored to get an even distribution around zero. Such a distribution improves the accuracy and performance of any algorithm that includes the square root. In the Harmonized Parabolic Synthesis methodology the accuracy is fine tuned by using intervals. The accuracy increases with the number of intervals and for single-precision floating-point square root, 64 intervals are sufficient. This method can be used for computing unary operations such as trigonometric functions, logarithms, and square root as well as binary functions such as division [7], [8], [13].

We have used the square root hardware, that is implemented in this paper, in an accelerator utilized for accelerating QR decomposition. The accelerator uses single-precision floating-point numbers and it is integrated to a RISC-V [14] core to build a heterogeneous tile [15]. It performs the square root operation on single-precision floating-point numbers represented in IEEE-754 standard format.

The rest of the paper is structured as follows: Section II provides the necessary background knowledge to understand the rest of the paper. Section III presents other related works on square root operation. Section IV goes through the methodology whereas Section V gives the details of the implementation on FPGAs. Results of the implementations are presented in Section VI with a discussion. Section VII finalizes the paper with concluding remarks.

II. BACKGROUND

In this section, we introduce the IEEE standard for floating-point number representation and give a brief overview of the square root methods.

A. Binary32 - Single-precision floating-point numbers

There are many floating-point number representations, however, the IEEE Standard for Floating-Point Arithmetic (IEEE-754) [6] is the most common one. This standard defines the representation of floating-point numbers and the arithmetic operations. In this paper, we focus on binary32 numbers, which are more commonly known as single-precision floating-point numbers.

The IEEE 754 standard specifies a binary32 (single-precision floating-point) number as having:

- Sign bit s : 1 bit
- Exponent width e : 8 bits
- Significand (mantissa) m : 24 bits (23 explicitly stored)

as illustrated in Figure 1.

Fig. 1. Single-Precision (binary32) floating-point representation in IEEE-754 format.

The sign bit determines the sign of the number. The exponent is an 8-bit unsigned integer and biased by +127. The true mantissa consists of 23 visible bits to the right of the binary-point and 1 invisible leading bit to the left of binary-point which is 1 unless the exponent is 0. The real value is calculated with the following formula:

$$(-1)^s \times (1 + m) \times 2^{e-127} \quad (1)$$

The exponent ranges from -126 to 127 because -127 (all zeros) and 128 (all ones) are reserved and indicate special cases. This format has the range of $\pm 3.4 \cdot 10^{38}$. The smallest positive normal number is $1.17 \cdot 10^{-38}$.

B. Floating-point square root

There are several algorithms for calculating the square root of a number dating back to Babylonians. Some of these algorithms are the Babylonian method [16], Bakhshali method [17], Newton-Raphson method [18] and, Taylor series expansion [19]. These methods are iterative methods and generally yield approximate results, but can be made arbitrarily precise by increasing the number of calculation steps. Usually, the square root algorithms require an initial seed value. If the initial seed value is far away from the actual square root, the algorithm will be slowed down. It is therefore useful to have a rough estimate, which may be inaccurate but easy to calculate. Our method uses a set of estimates in look-up tables and utilizes auxiliary functions to increase the accuracy of the estimations.

III. RELATED WORKS

There are several other studies aimed at implementing square root operation efficiently targeting FPGAs. We will mention a few of these studies, which use single-precision floating-point units. The related FPGA implementations usually use older FPGA platforms. Hence it is difficult to make a very accurate comparison between our implementation and the others. However, we still compare resource usage and clock frequency numbers to provide an estimation to the reader.

Suresh et al. [20] and Patil et al. [21] use a non-restoring iterative approach to implement floating-point square root operation on FPGAs and ASICs. In iterative approaches, the amount of resources and clock cycles (latency) increase rapidly

with the word size of the input. However, our approach requires fewer resources and cycles to produce a result. For instance, to produce a single result, the mentioned implementations require 40 and 31 cycles, while achieving 205 and 240 MHz clock frequencies respectively.

Dinechin et al. [22] use a multiplicative (polynomial approximation) method to implement square root operation with different precisions on FPGAs. Since our method is multiplicative as well, their results are close to our results, however, they claim that the cost of correct round is rather high. Hence our resource usage results are lower than what they achieve for correctly rounded, single-precision floating-point results.

Xilinx uses CORDIC method for their square root IP [23]. The achieved clock frequency is higher than what we achieve due to not using DSP and BRAM blocks and consequently having shorter pipeline stages. However, our implementation outperforms the IP in resource usage and latency.

Hasnat et al. [24] develop an FPGA implementation that calculates single-precision floating-point square root and inverse square root simultaneously with Quake's algorithm [25] modified using Newton Raphson method. They achieve quite low numbers for resource usage on Virtex 5, however, they do not support pipelining and do not provide the number for DSP usage. They use 7 multipliers and each seems to have 23 bits inputs. Thus we assume 2 DSPs are used for each multiplier.

The novelty of this work is the usage of the Harmonized Parabolic Synthesis method for performing square root on single-precision floating-point numbers. The hardware implementation uses three look-up tables, 5 integer multipliers and 7 integer adders, all with different word lengths. The look-up tables consist of 64 words with 28, 22 and 16 bits of word length respectively. These are significantly smaller when compared to other table look-up methods for the same accuracy. Additionally, the other works neglect the error distribution aspect, which can cause trouble due to accumulated errors when the calculation is part of a larger algorithm. Our methodology is able to provide a normal distribution of the error around zero to avoid possible accumulated error problem.

IV. METHODOLOGY

In this section we will go through a brief description of the mathematics behind the proposed method [8], how to apply it to IEEE-754 single-precision floating-point numbers and the accuracy and error behavior of the method.

A. Approximation

The proposed method performs a multiplicative approximation of the square root function and it can be classified as table look-up with auxiliary functions. In order to facilitate the calculations, the range of input and output numbers are constrained to $[0,1]$.

The approximation is calculated as the product of two sub-functions; $s1$ and $s2$. The first sub-function is a second order polynomial, whereas the second sub-function is a second order interpolation. The first sub-function roughly approximates the original function (square root) while the second sub-function

improves the approximation. The first sub-function $s_1(x)$ is defined as:

$$s_1(x) = x + c_1(x - x^2), \quad (2)$$

where c_1 is a coefficient. The second sub-function $s_{2,i}(x)$ is a second degree interpolation of a help function $f_{help}(x)$, defined as:

$$f_{help}(x) = f_{org}(x)/s_1(x), \quad (3)$$

where f_{org} is the original function. Splitting the help function into N intervals provides a set of equations shown below:

$$s_{2,i}(x) = l_{2,i} + j_{2,i}x_w - c_{2,i}x_w^2, \quad (4)$$

where $0 \leq i < N$, $w = \log(N)$ and x_w is fractional part of $2^w x$.

B. Application of the Approximation

In order to facilitate the usage of this method on floating-point numbers, two more steps are added. Thus, the final method consists of three steps namely pre-processing, processing and post-processing as seen in Fig. 2.

Fig. 2. The steps of the square root method.

In the pre-processing step the exponent is divided by two to get the square root as follows:

$$\text{sqrt}(2^e \times M) = 2^{e/2} \times \text{sqrt}(M), \quad (5)$$

where e is the unbiased exponent and M is the mantissa with the hidden bit. If the exponent is odd then it is reduced by 1 to get an integer after the division. In this case, the mantissa is multiplied by 2 to compensate the reduction in the exponent. Initially the input mantissa (including the hidden bit) has a value between 1 and 2. After the adjustments on the exponent and the mantissa the new range becomes $[1, 4)$. In order to move the range of the mantissa to $[0, 1)$, which is required by the approximation method, the mantissa is reduced by 1 and then divided by 3 as seen in Figure 3 (we perform some bit tweaking in the circuit to reduce the resource usage). These operations correspond to converting the number to base 4 [26].

In the processing step, the approximation methods (the parabolic synthesis and second degree interpolation) are applied to the mantissa of the input to approximate the square root. The result of this step is the product of the sub-functions.

In the post-processing step, the exponent is transformed back into IEEE 754 format by adding 127 to its value and the final result is produced. A detailed diagram of the method is given in Figure 4. The re-normalization corresponds to pre-processing, whereas approximation is the processing step and the operation on the exponent is the post-processing step.

C. Accuracy and Error Behavior

In total, the required number of operations are 5 integer multiplications, 7 integer additions and 1 shift operation. The number of operations do not change with the size of the input value. The same method can be used for arbitrary number of accuracy bits. The size of the operators and look-up tables would increase in case of wider inputs and higher accuracy. Intermediate and final results of the operations are truncated, which means they are rounded to zero to get the smallest error when compared to other rounding methods defined by the IEEE binary32 standards such as 'round to $+\infty$ ' or 'round to nearest' [6]. Hence 'round to zero' and naturally 'round to $-\infty$ ' are the only supported rounding methods.

The square root result is tested for $2^{23}(8,388,608)$ inputs (every possible mantissa value) twice; once for odd exponent case and once for even exponent case. The rounding errors for the tested inputs are smaller than 1 ULP (unit in the last place) [27], that is used as a measure of accuracy in numeric calculations and calculated as follows:

$$ULP(x) = \epsilon \times 2^{e'}, \quad (6)$$

where x is the result in single-precision floating-point format, ϵ is machine epsilon and e' is the unbiased exponent of x .

When performing computations, the accumulated error in the computations can corrupt the result rapidly. When comparing different approximation method, it should be noted that, to achieve the same performance with the same algorithm, different methods may require different precisions. Approximation methods should therefore be compared in the same context. As an illustration, a comparison is performed between typical distributions of the error of the CORDIC methodology and the Harmonized Parabolic Synthesis method. In Figure 5 the distributions of the error of an implementation using the CORDIC method is shown.

As can be seen in Figure 5 the error is not evenly distributed around zero, which will have devastating effect on the problems with accumulated error. The structure of the CORDIC algorithm prevents any significant improvement to the distribution of the error. As shown in [28], CORDIC requires several extra bits of accuracy in order to cope with the error in the approximation.

A significant advantage with the Harmonized Parabolic Synthesis methodology is that the distribution of the error of the approximation can be fully controlled [8]. To minimize the scattering effects on the distribution of the error it is important that it is evenly distributed around zero and with an advantageous shape of the distribution. Because of the central limit theorem, a useful distribution of the error is the normal distribution. In the most general form of the theorem and under some conditions (which include finite variance) it states that averages of random variables independently drawn from independent distributions converge in distribution to the normal, that is, become normally distributed when the number of random variables is sufficiently large [29]. As shown in Figure 6, it is possible with the Harmonized Parabolic Synthesis method to accomplish the desirable normal distributed error.

Fig. 3. Overview of the steps of the square root method implemented as 5 stages. This implementation supports normalized positive numbers. The vertical thick lines represent the stage registers. (M includes the hidden bit that is 1)

Fig. 4. The diagram of the method being applied to IEEE-754, single-precision floating-point numbers.

Fig. 5. A typical error distribution for the CORDIC method.

Fig. 6. A typical error distribution of the Harmonized Parabolic Synthesis method.

V. IMPLEMENTATION DETAILS

The square root hardware is implemented in different hardware description languages such as Chisel, VHDL and Verilog. The Chisel implementation is converted to Verilog to be synthesized via Xilinx tools. All implementations result in the same hardware and resource usage and support 3 out of 4 special input values defined by the IEEE binary32 format. The supported special input values are **zero**, $\pm$ **infinity** and, **NaN** (Not a Number). Denormalized numbers are currently not supported as we do not need them, however they can be easily supported in the future.

Two different implementations of the floating-point square root hardware are presented in this paper. The first implementation consists of a single stage and produces the output in 1 cycle, whereas the second implementation is pipelined and consists of 5 stages. The pipelined implementation produces the first output in 5 cycles and when the pipeline is filled 1 output per cycle is produced. Figure 3 shows the implementation

with 5 stages. Each blue (relatively large) box corresponds to a step in the method. The vertical thick lines, which are separating the stages, represent the stage registers.

On chip DSP blocks, which have 27×18 input width in Ultrascale FPGAs and 25×18 on Virtex-7 FPGAs, are utilized for the multiplication operations. However, most of the multiplications are performed on inputs, which have longer bit-width than the width supported by the DSP blocks. Therefore, these operations are performed on more than a single DSP block that are cascaded.

The coefficient sizes are 28 bits, 22 bits, and 16 bits. Since 64 intervals are sufficient/used for performing square root on single-precision floating-point numbers, we need to store 64 values for each coefficient. The total amount of memory required for storing the coefficients become $64 \times (28 + 22 + 16) = 4224$ bits or 528 bytes.

VI. RESULTS

In this section, we provide post-implementation resource utilization and timing results of our square root hardware implementations and compare them to the results of several related works. The FPGA synthesis is performed by Xilinx tools. The target platforms are Zybo Zynq-7000 board with Virtex-7 series FPGA and Xilinx VCU108 evaluation kit including Virtex UltraScale XCVU095-2FFVA2104E FPGA. Zybo is a low-end board and VCU108 is a high-end board.

Two different implementations of the square root hardware with different number of stages have been evaluated. The first implementation is fully combinational (consisting of a single stage) whereas the second implementation is pipelined (divided into 5 stages). The implementations have different results in terms of clock frequency and resource usage. Maximum frequencies are measured using the out-of-context flow to synthesize and implement the methods in isolation. This ensures that the design is not distorted due to routing to device pins. These results are presented in Table I. In the combinational implementation no registers are used, however, in the pipelined implementation stage registers are used.

TABLE I
COMPARISON OF DIFFERENT IMPLEMENTATIONS OF SINGLE-PRECISION
FLOATING-POINT SQUARE ROOT

	Zybo		VCU108	
	5 Stages	1 Stage	5 Stages	1 Stage
LUT	336	401	249	289
FF	205	0	172	0
BRAM	1.5	0	1.5	0
DSP	9	9	7	7
Freq	135 MHz	29 MHz	240 MHz	60 MHz

UltraScale FPGA has a higher technology than the Virtex-7 FPGA. Therefore the critical path becomes shorter and the clock frequency gets almost doubled while moving from Virtex-7 to Ultrascale. Due to the differences in the configurable logic block (CLB) structures, the register and look-up-table (LUT) numbers vary between the FPGAs. The block

ram structures are the same, thus the usage of BRAMs do not change. However, the DSP blocks in the Ultrascale FPGA have slightly larger input width and this causes utilization of fewer DSP blocks.

When compared to Xilinx square root intellectual property (IP) [23], [30], which is implemented via CORDIC method [9], [10], our implementation on VCU108 uses fewer LUTs and FFs. However, the Xilinx IP achieves higher clock frequency due to not utilizing any BRAMs or DSPs and dividing the hardware into shorter pipeline stages. In total they have 25 stages, whereas we have only 5. In terms of latency, our implementation shows better performance due to having fewer pipeline stages. The latency of our implementation is less than half the latency of Xilinx IP. This comparison is given in Table II along with other studies.

The implementation of Dinechin et al. [22] uses 241 slices, which is approximately 1k LUTs and 1k FFs. They also make use of on board DSP units and block RAMs. The frequency they achieve is more or less the same when compared to our implementation on Ultrascale FPGA, however, their latency is more than double of our latency due to requiring more cycles to produce a result. Perhaps they could achieve higher frequency with a newer FPGA, however it is difficult to determine what the resource usage would be.

Suresh et al. [20] do not make use of the DSPs or BRAMs as they are using a non-restoring iterative method that produces a result in 40 cycles. The resource usage of their implementation is significantly higher than our implementations on both Virtex-7 and Ultrascale FPGAs. The latency they get is approximately tenfold higher than our implementation on Ultrascale and fivefold on Virtex-7. This is due to the large number of cycles they need for producing a result.

Hasnat et al. [24] uses 199 LUTs and 24 FFs which are quite low. However, we assume they use 14 DSPs which is above our numbers. Additionally, their latency is higher and throughput is lower than what we achieve.

We should note that applying a multiplicative method to compute the square root on an FPGA makes use of the on-chip DSP blocks, however, these blocks limit the maximum clock frequency by preventing the stages to be shorter. Especially if the inputs are too large, several DSP blocks can be concatenated and the stages can become even longer.

VII. CONCLUSION

We have presented a novel method for implementing single-precision floating-point square root operation based on combination of two methods namely parabolic synthesis and second degree interpolation. We have implemented single-cycle and pipelined versions of the method supporting the IEEE single-precision floating-point standard and synthesized on two different FPGA boards. We compared the resource utilization and timing results of our implementations with the results of other studies and a Xilinx IP.

We are aware that comparison between works with different FPGAs is not very accurate. We do the comparison to give an opinion to the reader. Our implementations are smaller

TABLE II
COMPARISON OF OUR IMPLEMENTATION WITH SEVERAL OTHER IMPLEMENTATIONS. *VIRTEX 5 SLICES INCLUDE 4 FF AND 4 LUT

	Our imp	Xilinx IP [23], [30]	Dinechin et al. [22]	Suresh et al. [20]	Hasnat et al. [24]
Platform	Virtex Ultrascale	Virtex Ultrascale	Virtex 5	Virtex 6	Virtex 5
LUT	249	448	241 slices*	819	199
FF	172	786	241 slices*	531	24
DSP	7	0	5	0	14
BRAM	1.5	0	2	0	-
Frequency	240	568	237	205	199
Latency (cycles)	5	25	12	40	12
Latency (ns)	20.8	44.0	50.6	195.1	60.3-
Throughput	1 FLOP/cycle	1 FLOP/cycle	N/A	N/A	1 FLOP/12 cycles

than most of the other works with the exception of DSP and BRAM usage. In terms of latency, our designs outperform the other works. Therefore, they are suitable for high performance embedded systems. They can be used either standalone or integrated to another system. Both DSP and BRAM usages are open to optimizations as the data can easily fit into a single BRAM and number of the DSP blocks can be decreased by further optimizations on the sizes of intermediate results.

The error distribution is neglected in all studies. However, works based on CORDIC show an error distribution away from zero. Hence the errors can accumulate and cause accuracy problems in the final result. Our method is able to perform a normal distribution of the errors evenly around zero, which prevents accumulated error problems.

REFERENCES

- [1] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.
- [2] S. Savas, S. Raase, E. Gebrewahid, Z. Ul-Abdin, and T. Nordström, "Dataflow implementation of qr decomposition on a manycore," in *Proceedings of the Third ACM International Workshop on Many-core Embedded Systems*. ACM, 2016, pp. 26–30.
- [3] S. Savas, "Linear algebra for array signal processing on a massively parallel dataflow architecture," 2009, Thesis report, School of Information Technology, Halmstad University.
- [4] P. Luethi, A. Burg, S. Haene, D. Perels, N. Felber, and W. Fichtner, "Vlsi implementation of a high-speed iterative sorted mmse qr decomposition," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. IEEE, 2007, pp. 1421–1424.
- [5] E. Larsson, O. Edfors, F. Tufvesson, and T. Marzetta, "Massive mimo for next generation wireless systems," *Communications Magazine, IEEE*, vol. 52, no. 2, pp. 186–195, 2014.
- [6] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*, Aug. 2008.
- [7] E. Hertz, B. Svensson, and P. Nilsson, "Combining the parabolic synthesis methodology with second-degree interpolation," *Microprocessors and Microsystems*, vol. 42, pp. 142–155, 2016.
- [8] E. Hertz, J. Lai, B. Svensson, and P. Nilsson, "The harmonized parabolic synthesis methodology for hardware efficient function generation with full error control," *Journal of Signal Processing Systems*, pp. 1–15, 2017.
- [9] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.
- [10] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 191–200.
- [11] I. Newton, "Methodus fluxionum et serierum infinitarum," 1664-1671.
- [12] J. Raphson, "Analysis aequationum universalis," 1690.
- [13] S. Savas, E. Hertz, T. Nordström, and Z. Ul-Abdin, "Efficient single-precision floating-point division using harmonized parabolic synthesis," in *VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*. IEEE, 2017, pp. 110–115.
- [14] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.
- [15] S. Savas, Z. Ul-Abdin, and T. Nordström, "Designing domain-specific heterogeneous architectures from dataflow programs," *Computers*, vol. 7, no. 2, p. 27, 2018.
- [16] O. Kosheleva, "Babylonian method of computing the square root: Justifications based on fuzzy techniques and on computational complexity," in *Fuzzy Information Processing Society, 2009. NAFIPS 2009. Annual Meeting of the North American*. IEEE, 2009, pp. 1–6.
- [17] M. Channabasappa, "(02) on the square root formula in the bakhshali manuscript," 1976.
- [18] M. A. Cornea-Hasegan, R. A. Golliver, and P. Markstein, "Correctness proofs outline for newton-raphson based floating-point divide and square root algorithms," in *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*. IEEE, 1999, pp. 96–105.
- [19] T.-J. Kwon and J. Draper, "Floating-point division and square root using a taylor-series expansion algorithm," *Microelectronics Journal*, vol. 40, no. 11, pp. 1601–1605, 2009.
- [20] S. Suresh, S. F. Beldianu, and S. G. Ziavras, "Fpga and asic square root designs for high performance and power efficiency," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE, 2013, pp. 269–272.
- [21] V. Patil, A. Raveendran, P. Sobha, A. D. Selvakumar, and D. Vivian, "Out of order floating point coprocessor for risc v isa," in *VLSI Design and Test (VDATE), 2015 19th International Symposium on*. IEEE, 2015, pp. 1–7.
- [22] F. De Dinechin, M. Joldes, B. Pasca, and G. Revy, "Multiplicative square root algorithms for fpgas," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 574–577.
- [23] (Xilinx Inc, 2017) Xilinx cordic v6.0 logi-core ip product guide. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf
- [24] A. Hasnat, T. Bhattacharyya, A. Dey, S. Halder, and D. Bhattacharjee, "A fast fpga based architecture for computation of square root and inverse square root," in *2017 Devices for Integrated Circuit (DevIC)*. IEEE, 2017, pp. 383–387.
- [25] D. Eberly, "Fast inverse square root, geometric tools, llc (2010)."
- [26] E. Hertz, "Methodologies for approximation of unary functions and their implementation in hardware," Ph.D. dissertation, Halmstad University Press, 2016.
- [27] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.
- [28] A. Omondi, *Computer-Hardware Evaluation of Mathematical Functions*. World Scientific, 2015.
- [29] R. Durrett, *Probability: theory and examples*. Cambridge university press, 2010.
- [30] (Xilinx Inc, 2017) Performance and resource utilization for floating-point v7.1. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html